Verification of Timed Erlang Programs Using McErlang

Clara Earle, Lars-Åke Fredlund

Verification of Timed Erlang Programs using McErlang*

Clara Benac Earle[1] and Lars-Åke Fredlund[1]

Babel group, DLSIIS, Facultad de Informática, Universidad Politécnica de Madrid
{cbenac,lfredlund}@fi.upm.es

Abstract. There is a large number of works that apply model checking to timed *specifications*, however, there are far fewer attempts at model checking concurrent *programs* for which correct timed behaviour is crucial. In this work we explore the formal verification of timed programs written in the Erlang concurrent programming language, in its full complexity, using the McErlang model checker.
We have extended the McErlang model checker with a timed semantics, similar to the timed semantics Lamport has developed for TLA and TLC, but with a few notable differences. In the paper we present the resulting semantics, its implementation in McErlang, and evaluate it using a number of examples. Among the examples is a process supervision component for controlling the processes in an Erlang application, which provides fault-tolerance.

1 Introduction

Timed semantics for concurrent formalisms is by now a very well-studied field. In the field of real-time semantics a very successful technique for specifying and verifying systems are the timed automata [1]. Similarly there exist numerous discrete-timed specification formalisms, or timed formalisms for which a discrete time domain has been extensively studied, e.g., in process algebra for Timed CSP [2], TCCS [3], TPCCS [4], LOTOS [5], to mention but a few.

Uppaal [6] is the currently most well-known model checker for real-time systems. It provides an integrated tool environment for modeling, validation and verification of real-time systems modeled as networks of timed automata, extended with data types (bounded integers, arrays etc.).

While tools and specification formalisms like Uppaal are quite successful at real-time verification, there is still a need to reason about timed behaviour in other specification and programming languages, using dedicated model checkers. However, many of these model checkers do not implement tailored real-time verification algorithms like Uppaal. Rather, they are used to check timed behaviour by discretizing time, and by using normal model checking algorithms for the LTL

* This work has been partially supported by the following projects: DESAFIOS10 (TIN2009-14599-C03-00), PROMETIDOS (P2009/TIC-1465), pSAFECER (GA 269265) and nSAFECER (GA 295373).

and CTL logics. The SPIN [7] model checker is perhaps the most well-known explicit-state model checker that follows this approach.

In recent years the approach of checking timed behaviour using "untimed" model checkers has received renewed attention, with the publication of Lamport's article [8] on real-time model checking using the TLC model checker. Other recent works along the same lines include [9] and [10].

In this work we address the verification of timed programs written in the Erlang functional concurrent programming language using the McErlang model checker [11]. The approach taken is similar to Lamport's approach to real-time model checking in [8], and also inspired by the timed automata framework, but with a few notable differences. As in Lamport's work, the transitional semantics is defined over a global system state. In contrast to that article there is no explicit clock tick (the minimum observable "time quanta"), rather the tick is derived from the time constraints for the processes in the system, and the granularity of the tick can even vary through the lifetime of the verified system. Moreover, there is no explicit clock process which is responsible for the progress of time. As demonstrated, such processes can be programmed, and if desired, included in a verification.

In related work for Erlang, Guo et.al. [12], verify untimed Erlang programs by translating Erlang into the μCRL process algebra. However, the translation addresses only the high-level concurrency libraries of Erlang. In [13], Guo and Derrick consider the verification of timed Erlang/OTP components by means of translating Erlang into μCRL, however without considering the problem of generating finite models (state graphs).

In Sect. 2 and 3 we provide a brief introduction to the Erlang programming language and the McErlang model checker. Then, in Sect. 4, we provide an intuition for the timed semantics for Erlang, and Sect. 5 defines a high-level formal semantics. In Sect. 6 we evaluate the efficiency of the timed semantics; Sect. 7 summarizes the results.

2 Erlang

Erlang [14, 15] is a functional concurrent programming language created by the Ericsson company in the 1980s. Ericsson is still maintaining the main Erlang implementation, but it is available as open source since 1998. The chief strength of the language is that it provides excellent support for concurrency, distribution and fault tolerance on top of a dynamically typed and strictly evaluated functional programming language. Concurrency is achieved by lightweight processes communicating through asynchronous message passing. Although Erlang is not a new language, it has experienced considerable growth in users in recent years. This is due in most part to its focus on message passing instead of variable sharing as the main communication mechanism, which enables programmers to write robust and clean code for modern multiprocessor and distributed systems.

Today Erlang is used by Ericsson and many other companies (T-Mobile (UK), and many smaller start-up companies such as e.g. LambdaStream in Spain and

Klarna in Sweden) to develop industrial applications, often implementing crucial internet server-side applications. Examples include a high-speed ATM switch developed at Ericsson with over a million lines of Erlang code which had to meet very challenging requirements on software reliability and overall system availability [16, 17], parts of Facebook chat, Apache CouchDB – a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API, etc.

Handling a large number of processes easily turns into an unmanageable task, and therefore Erlang programmers mostly work with higher-level language components. The OTP component library is the most used, it offers design patterns such as: a generic server component (for client-server communication), a finite state machine component, generic TCP/IP communication, and a supervisor component for easy structuring of fault-tolerant systems.

2.1 Handling Time in Erlang

As Erlang is relatively well known, we will just describe the main language features which concern timing, i.e., the receive statement and timestamps.

The receive Statement. The basic mechanism for handling time dependent behaviour in Erlang is the timeout clause of a receive statement:

```
receive
  Pat1 when Guard1 -> Expr1;
  ...
  PatN when GuardN -> ExprN
after Deadline -> TimeoutExpr
end
```

The intuitive semantics of the receive statement is as follows. If a message matches a pattern `PatI`, and the guard `GuardI` (which may contain variables bound by the match) evaluates to true (and moreover no earlier pattern `Patj` matches, or the guard `GuardJ` does not evaluate to true), the message is removed from the mailbox and evaluation continues with expression `ExprI` under the matching binding.

Concretely, the oldest message in the process mailbox is first matched against the clauses according to the above procedure. If no pattern and guard match this message, the same sequence of tests continues with the second oldest message, and so on. If no message matches, the process waits for the reception of a matching message for *at least* `Deadline` milliseconds, until it times out and starts executing the expression `TimeoutExpr`.

A zero deadline corresponds to the case when, if no matching message is in the mailbox, the timeout can happen at once. The special atom `infinity` may also be used as a time deadline, signifying waiting forever without timing out.

Timestamps. The API call `now()` returns the time elapsed since 00:00 GMT, January 1, 1970 as a tuple `{MegaSeconds,Seconds,MicroSeconds}`.

3 McErlang

McErlang [11, 18] is an explicit-state model checker for programs written in Erlang. Similarly to most explicit state model checkers, McErlang checks concurrent programs against specifications in full linear temporal logic (LTL) using on-the-fly state space exploration algorithms.

The main idea behind the design of McErlang is to re-use as much of the normal Erlang language implementation as possible, but adding a model checking capability. To achieve this, the tool replaces the part of the Erlang runtime system which implements concurrency and message passing, while still using the runtime system for the evaluation of the sequential part of the input programs. McErlang has built-in support for some Erlang OTP component behaviours that are used in almost all serious Erlang programs such as the supervisor component (for implementing fault-tolerant applications) and the generic server component (implementing a client–server component). The presence of such high-level components in the model checker significantly reduces the gap between original program and the verifiable model, compared to other model checkers.

McErlang has been used in several complex case studies: in the model checking of a Video-on-Demand-server [19], in the verification of agent based RoboCup teams [20], and for the verification of an industrial Erlang process supervision component [21]. The timed extension of McErlang described in this paper is available at GitHub [22].

4 A Timed Extension

In the following subsections we first provide an intuition for the untimed Erlang semantics in McErlang, then introduce the timed extension and timestamps.

4.1 An Untimed Semantics

Previously, there was only an untimed semantics implemented in McErlang. In the untimed semantics, if a receive statement cannot be executed because there is no matching message in the process mailbox, the timeout is enabled (unless the deadline is `infinity`). However, another process can send a receivable message to the process and thus disable the timeout. This corresponds to treating timeouts as nondeterministic choices. As an example, consider code below.

```
P1 = spawn(fun () -> receive Msg -> ok
                     after 1000 -> bad
                     end
           end),
spawn(fun () -> P1!hello end).
```

Two processes are spawned, the first (`P1`) waiting to receive a message, and timing out after one second (1000 milliseconds) if no message can be received, and the second process sending the message `hello` to the first.

The state graph of the above program as generated by McErlang is depicted in Fig. 1 below. Note that only side effects are depicted; as the receive statement is not considered a side effect it is not shown. We colour the states where the `hello` message was received light grey, whereas states where a timeout occurred first are grey, and the arrow of the timeout transition is bold. Clearly there is a race condition in the program: if the message `hello` is sent first from `P1` to the second process a timeout never happens. Alternatively the timeout can happen first; in this case the message is sent anyway but is never received.

Non-Timed Actions are Infinitely Fast. A semantics option, implemented in McErlang, provides a semantics where non-timeout actions are infinitely fast compared to timeouts, i.e., always giving precedence to non-timeout actions. In practice this turns out to be a useful abstraction in many scenarios where the actions of the verified program can be safely assumed to be infinitely fast compared to timed (external) actions.

The state graph for the above program, when this option is enabled, is shown in Fig. 2. Since a timeout is infinitely slow compared to other actions, the timeout never happens since it is disabled by the reception of the `hello` message.

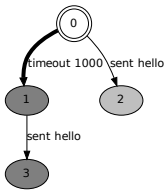

Fig. 1: State graph with no precedence

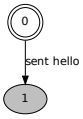

Fig. 2: State graph with precedence

4.2 Adding Explicit Time

The main changes needed in McErlang to implement a timed semantics are to record the current time in the state representation of a running program, and to modify the behaviour of the receive statement in the model checker so that when handling timeouts, the current time is taken into account.

To keep compatibility with normal Erlang code we let the current time be a tuple `{MegaSeconds,Seconds,MicroSeconds}`, and its initial value is `{0,0,0}`.

Clearly the presence of a non-infinity timeout clause in a receive statement specifies a *minimum* waiting period until a timeout happens. In the Erlang documentation there is of course no guarantee for exactly when, after a timer has elapsed, the corresponding timeout happens (as it depends on the operating system, the hardware, etc). However, in this work, as is usual for timed calculi,

we also want to be able to specify a *maximum* waiting period until a timeout happens (a notion sometimes called *urgency* in timed calculi).

To specify the urgency of a state the function `mce_erl:urgent(MaximumWait)` is provided. The parameter `MaximumWait` specifies the maximum number of milliseconds the process can remain in the current state, if it has transitions enabled.

Moreover we reinterpret the notion of an infinitely fast computation (i.e., where normal actions are infinitely fast compared to timeouts) as one where every state has an associated implicit call to `mce_erl:urgent(0)` signifying that no time can pass if the state has a transition enabled.

For instance, we can add the line `mce_erl:urgent(1500)` to the running program example (see Fig. 3) to force a timeout to happen before some moment in time. The first process will now wait between 1 and 1.5 seconds for a message to arrive before timing out. Since we have not specified when the second process sends a message both possibilities (timeout or no timeout) remain possible. Its state graph is depicted in Fig. 4; each state is labelled by the current time value in milliseconds.

```
P1 =
  spawn
  (fun () ->
     mce_erl:urgent(1500),
     receive Msg -> ok
     after 1000 -> bad
     end
   end),
spawn(fun () -> P1!hello end).
```

Fig. 3: Program 2

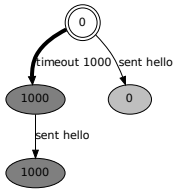

Fig. 4: State graph with urgency and a clock

It may be surprising to see that time does not progress in Fig. 4 after the timeout, and that the timeout occurs exactly at the earliest possible moment. This is because the semantics discretizes the progress of time: time makes a discrete jump between two consecutive time values. So what are the consecutive time values? In normal discrete-time semantics there is often an implicit clock, with a *tick* value which defines the minimum distance between two time values:

$$t_0 = 0 \qquad t_1 = 1 * tick \qquad t_2 = 2 * tick \qquad \cdots$$

where the t_i's are the time values at different states. In our semantics, in contrast, there is no implicit clock process nor a hard-wired tick clock value increment. Rather, each timeout clause in a receive statement represents a clock tick.

However, we can easily implement an *explicit* clock process, which constantly increments the time value with a tick increment as seen in Fig. 5. Note that if we add an unbounded clock process to the program in Fig. 3, then its state

```
clock(Tick) ->
  mce_erl:urgent(0),
  receive
  after Tick -> clock(Tick)
  end.
```

Fig. 5: An unbounded clock

```
clock(Tick,0) -> ok;
clock(Tick,N) when N>0 ->
  mce_erl:urgent(0),
  receive
  after Tick ->
    clock(Tick,N-Tick)
  end.
```

Fig. 6: A bounded clock

graph becomes infinite. To obtain a finite state graph for now (in Sect. 5.1 we overcome this restriction) the bounded clock in Fig. 6 is used instead.

The state graph of program 2, where we add a clock process with a 500ms tick and a duration of 2500ms with the call `spawn(fun () ->clock(500,2500) end)`, is shown in Fig. 7. The timeout transitions in the graph are either timeouts by a process (corresponding to bold arrows as usual) or timeouts by the clock process. In the graph we can see that the timeout behaviour (the bold transition) is enabled after 500ms, and stays enabled until after 1500ms. Note also the white state labelled with 1500 (ms); since the timeout transition with value 0 is urgent, the clock process cannot make a transition which increases time.

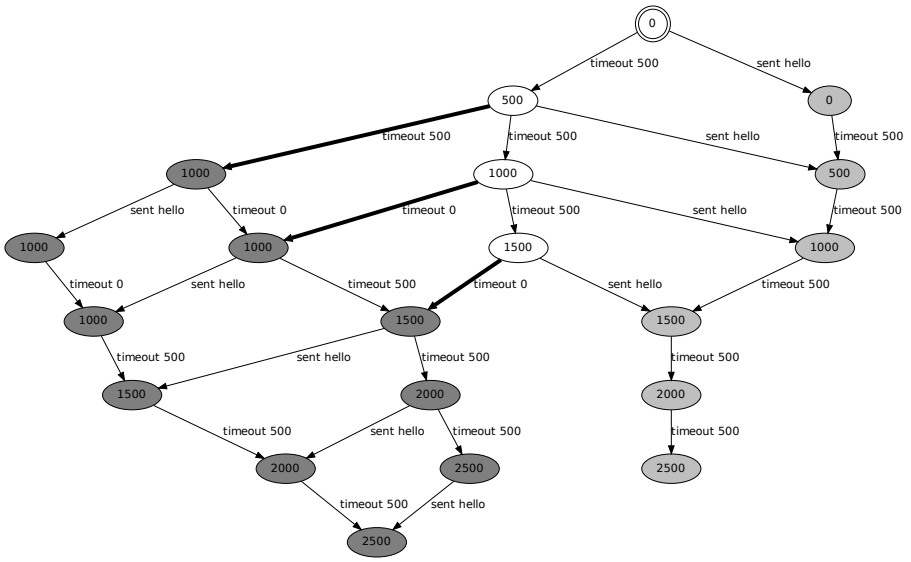

Fig. 7: State graph with an explicit bounded clock process

4.3 Supporting Timestamps

Clocks have an important role to play in decidable real-time calculi, recording the moment an event takes place, relative to other clocks and the general progress of time. In Erlang the function `now()` returns the time elapsed since 00:00 GMT, January 1, 1970 as a tuple `{MegaSeconds,Seconds,MicroSeconds}`. Most Erlang programs that handle time simply call `now`, store the result somewhere, and later compare the old value with the current time. This programming discipline is reminiscent of the use of clocks in timed automata formalisms.

The discussion on how to obtain finite models with time is deferred to Sect. 5.1; however, a crucial requirement is that only a finite number of values returned from calls to `now()` are "alive" (i.e., accessible from some program variable) in any state. It would be possible to tailor a static analysis to track calls to `now()`, and the flow of the resulting values. Instead we provide the programmer with a new API to obtain time stamps, and to directly manipulate the lifetime of timestamps. Direct calls to `now()` are forbidden. In general adapting a program to use this new API is trivial; an example is provided below, and Sect. 6.2 contains a further discussion.

The new API located in the module `mce_erl_time` has the following functions:

`now()` – returns the current time
`nowRef()` – stores the current time in a clock reference
`was(Ref)` – returns the time stored in a clock reference
`forget(Ref)` – explicitly destroys an old clock reference

There are a number of restrictions on the use of this API. First, times obtained from calls to `now()` may never be remembered by the program, but only used in comparisons against previously recorded clocks. Moreover, for the soundness of model checking it is forbidden to compare a clock (or the current time) against an absolute time value, only relative comparisons are permitted, e.g., checking how much time has passed since some system event occurred. That is, it is not allowed to check whether `now()` returns some concrete date and time.

To illustrate the clock API we show below the coding of a fragment of the lamp example [23] of real-time calculi. A lamp is initially *off*, and when a button is pressed shines with *low* intensity. If a button is pressed again within 5 milliseconds, the lamp shines with *bright* intensity, otherwise if the new button press arrives later, the lamp is switched *off*. In the code fragment below we show the lamp controller, which receives button presses from a user, and operates the lamp hardware by sending messages to `PhysicalLamp`. The function `compareTimeStamps_ge(T1,T2)` returns true if `T1` is a later or identical timestamp as `T2`; `addTimeStamps(T1,T2)` computes a new time stamp which is the sum of its argument time stamps, and `milliSecondsToTimeStamp(N)` computes the time stamp corresponding to `N` milliseconds.

```
lamp(PhysicalLamp) ->
  receive
    press ->
      PressTime = mce_erl_time:nowRef(),
```

```
    PhysicalLamp!low,
    receive
      press ->
        case compareTimeStamps_ge
             (mce_erl_time:now(),
              addTimeStamps(milliSecondsToTimeStamp(5),
                            mce_erl_time:was(PressTime))) of
          true ->
            PhysicalLamp!off,
            mce_erl_time:forget(PressTime), ...;
          false ->
            PhysicalLamp!bright,
            mce_erl_time:forget(PressTime), ...
        end
    end
end.
```

5 A Semi-Formal Timed Semantics

In the following we describe how a timed semantics can be obtained by modifying an untimed semantics. We assume the presence of a non-hierarchical "global" structured operational semantics for Erlang states, which is the basis of the implementation of the McErlang model checker (unpublished work). For a non-global state, hierarchical, semantics of Erlang programs see [24, 25].

In the untimed "global" semantics, an Erlang state s is, informally, a tuple $\langle Nodes, Ether\rangle$ consisting of a set of nodes ($Nodes$) and a datastructure ($Ether$) storing the messages in transit between nodes. To obtain a timed Erlang state we add the current time $Time \equiv \langle MegaSeconds, Seconds, MicroSeconds\rangle$ and a set of clocks, i.e., tuples $\langle ClockId, Time\rangle$ created by calling `nowRef()`, to the tuple: $\langle Nodes, Ether, Time, Clocks\rangle$. A node $\langle Processes, Dictionary, Registry, Links, Monitors\rangle$ is a collection of processes, a node global variable store, a process registry (for associating symbolic names to processes), and process links and monitors (for handling fault tolerance). Finally a process $\langle Pid, Dictionary, Mailbox, Expr\rangle$ has a mailbox, a process dictionary (an imperative memory), a unique process identifier, and the currently executing expression $Expr$. The contents of a typical Erlang system state is depicted symbolically in Fig. 8. Solid lines depict inter-node message passing, dashed lines intra-node message passing.

An Erlang untimed action is, informally, a side effect (e.g., a message sent between two processes, registering a symbolic name for a process, etc) or a process internal action. To the untimed actions we add the timed actions corresponding to timeouts, which cause time to progress, and actions corresponding to creating and modifying clocks. We let α range over the actions.

Given that we can compute the untimed transition relation written $s \xrightarrow{\alpha} s'$, which is defined as an structural operational semantics, we obtain the timed transition relation $s \xrightarrow[pre_t]{\alpha} s'$ by copying the transition rules from the untimed

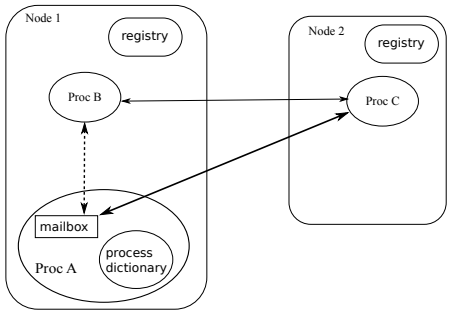

Fig. 8: An Erlang multi-node system

semantics *except* the rules for timeouts, and by adding a few transition rules concerning timeouts and clock handling to the timed semantics.

For timeout handling we add two rules:

$$\frac{\begin{array}{c} p \in \mathit{processes}(s) \qquad p.\mathit{expr} \equiv \texttt{receive}\ \mathit{clauses}\ \texttt{after}\ \mathit{deadline}\ \texttt{->}\ e\ \texttt{end} \\ \mathit{deadline} \neq \texttt{infinity} \qquad \mathit{absdeadline} = s.\mathit{time} + \mathit{deadline} \\ p' = p \text{ where } p'.\mathit{expr} = \texttt{absreceive}\ \mathit{clauses}\ \texttt{after}\ \mathit{absdeadline}\ \texttt{->}\ e\ \texttt{end} \end{array}}{s \xrightarrow[pre_t]{} s[p'/p]}$$

$$\frac{\begin{array}{c} p \in \mathit{processes}(s) \qquad p.\mathit{expr} \equiv \texttt{absreceive}\ \mathit{clauses}\ \texttt{after}\ \mathit{absdeadline}\ \texttt{->}\ e\ \texttt{end} \\ \neg \mathit{receivable}(p) \\ s' = s \text{ where } s'.\mathit{time} = \mathit{absdeadline} \qquad p' = p \text{ where } p'.\mathit{expr} = e \end{array}}{s \xrightarrow[pre_t]{timeout(absdeadline)} s'[p'/p]}$$

The first side effect free rule simply replaces a time relative deadline with an absolute deadline; to clarify the semantics we introduce the new synthetic keyword `absreceive` for such time absolute receive statements.

In the second rule, in a state s, if there is a process p which is executing a receive statement, and which cannot receive a message, then there is a transition labelled by the action $\mathit{timeout}(\mathit{deadline})$ to a new state where the current time has increased, and where the currently executing expression of p has been replaced. Similar transition rules are added for handling clocks.

Finally we constrain the resulting transition relation to enforce global urgency constraints on the progress of time, using the following high-level Erlang function which first calculates the transitions for a state using the function `transitions`, and then returns a reduced set of transitions compatible with the notion of urgency introduced in Sect. 4.2.

```
timeRestrict(State) ->
  Now = State#state.time,
```

```
  Transitions = transitions(State),
  timeRestrict(Transitions,Now,[],infinity,[]).

timeRestrict([],_,Untimed,_,Timed) -> Untimed++Timed;
timeRestrict(Transitions,Now,Untimed,MostUrgent,Timed) ->
  [Transition|Rest] = Transitions,
  MinWait = calculate_minwait(Transition),
  MaxWait = calculate_maxwait(Transition),
  if
    MinWait==infinity orelse MinWait > MostUrgent ->
      timeRestrict(Rest,Now,Untimed,MostUrgent,Timed);
    MaxWait==infinity andalso MinWait==Now ->
      NewNonTimed = [Transition|NonTimed],
      timeRestrict(Rest,Now,NewNonTimed,MostUrgent,Timed);
    MinWait =< MaxWait < MostUrgent ->
      NewTimed = [Transition|restrict(MaxWait,TimerEntries)],
      timeRestrict(Rest,Now,NonTimed,MaxWait,Newtimed);
    MinWait =< MostUrgent =< MaxWait ->
      NewTimed = [Transition|TimerEntries],
      timeRestrict(Rest,Now,NonTimed,MostUrgent,NewTimed)
  end.

restrict(MaxWait,[]) -> [];
restrict(MaxWait,[Transition|Rest]) ->
  MinWait = calculate_minwait(Transition),
  if
    MinWait =< MaxWait -> [Transition|restrict(MaxWait,Rest)];
    MinWait > MaxWait -> restrict(MaxWait,Rest)
  end.
```

The `timeRestrict` function is called with the following arguments: a list of transitions which is reduced, the current system time (`Now`), a list of untimed transitions (`Untimed`), the time when the most urgent transition seen so far must be taken (`MostUrgent`), and a list of transitions which are enabled to be executed sometime before the `MostUrgent` deadline. The function first classifies a timed transition: informally the minimum waiting period is the timeout value in a receive statement, whereas the maximum waiting period is the urgency (specified using a call to `mce_erl:urgent`). Having `infinity` as the wait limit signifies that the process will wait forever.

5.1 Finite Models

To obtain finite models, i.e., finite state graphs, for timed programs, we note that the actions of a timed program normally depends only on the passage time, not on the absolute value of the time parameter of the system state. Similarly, in the specification logic we make statements only about the relative value of clocks and the system time parameter. Thus, for any given system state there is typically an infinite number of equivalent states, which differ only in that

the time system parameter is distinct, but the relative values of clocks and the system time parameter is the same for all these "equivalent" states.

Thus, to obtain finite models, the obvious strategy is to normalize system states, and to generate the state space modulo such normalization. During state space generation, before adding a new node to the state graph, we should check whether its normalization (another state) is already in the graph. If it is, we do not need to consider the new state further. If it is not, we add the normalized state to the state graph, and continue exploring the behaviour of the new state. In [8] this procedure is referred to as model checking under symmetry.

It turns out to be trivial to add such a normalization to McErlang. The untimed McErlang tool already provided an "abstraction" feature, whereby a user-defined state abstraction function can be used to transform a state before storing it in the state table; exactly what is needed to implement time normalization. The normalization procedure modifies a state according to the following:

- The system time parameter is reset to `{0,0,0}`
- The values of clocks (created using `nowRef()`) are decreased by the old time
- Timeouts in (absolute) receive statements are decreased by the old time (but greater or equal to the new time)

As an example, the state graph for the program in Fig. 3, with the unbounded clock in Fig. 5, and using the above normalization, is depicted in Fig. 9. Note that states are no longer labeled by the current time, as the normalization collapses many states with distinct times into a single state.

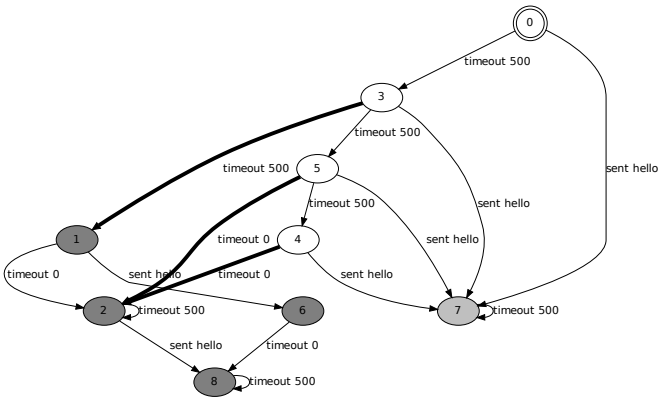

Fig. 9: State graph for the program in Fig. 4 with an explicit unbounded clock process, and finite model abstraction

Note that normalization does not guarantee finite state spaces. A program may for instance create an unbounded number of clock references, leading to an infinite state graph.

6 Experiments

To evaluate the resulting semantics and its implementation we below provide some initial benchmark results, and report on the challenges in obtaining a verifiable timed model from a crucial Erlang software component used in industry.

6.1 Efficiency

We evaluate the efficiency of the resulting implementation by checking Fischer's mutual exclusion algorithm [26]. Fig. 10 contains an implementation of the algorithm in Erlang. The global node dictionary extension of McErlang is used to implement reading and writing to the shared variable; see the functions `read` and `write`. The entering of the critical region of a process `Id` is indicated by a synthetic probe action `mce_erl:probe({enter,Id})`. The `latest(Tick,Wait,F)` function calls the function parameter `F` at most `Wait` milliseconds later; the time interval is partitioned into slices of maximum size `Tick` (the slices could be smaller if other clocks are defined).

We check mutual exclusion in a range of experiments characterized by the parameters of the `start(N,Tick,D,T)` function: `N` is the number of processes, `Tick` is the time tick, `D` is the maximum time to wait until writing to the shared variable, and `T` is the minimum time to wait until reading from the shared variable. Note that in this experiment we do not assume that internal actions are infinitely fast compared to timers.

Table 1: Execution times for Fischer's algorithm

N	Tick	D	T	Time (secs.)	Number of states
4	1	1	2	0.1s	2034
5	1	1	2	0.7s	13738
6	1	1	2	7.6s	89051
7	1	1	2	50.3s	580080
5	1	2	3	4.4s	81452
5	1	3	4	12.7s	268793
5	1	4	5	36.7s	704901
5	1	5	6	73.7s	1522179

As seen in Table 1 the size of the state space is exponential in the number of processes (`N`). In the last four rows the effect of an increase in the values of the timers is indicated. To verify the correctness of the algorithm a simple monitor checks that `enter` and `exit` probe actions strictly alternate. As expected, if writing is slower than reading, e.g., `D` $>$ `T`, the algorithm works correctly and otherwise a counterexample is quickly found. Overall, the size of state spaces and the execution times are reasonable.

```
start(N,Tick,D,T) ->
  write(0),
  lists:foreach
    (fun (Id) -> spawn(fun () -> idle(Id,Tick,D,T) end) end,
     lists:seq(1,N)).

idle(Id,Tick,D,T) ->
  case read() of
    0 -> set(Id,Tick,D,T);
    _ -> idle(Id,Tick,D,T)
  end.

set(Id,Tick,D,T) ->
  latest(Tick,D,fun () -> setting(Id,Tick,D,T) end).

setting(Id,Tick,D,T) ->
  write(Id),
  sleep(T),
  testing(Id,Tick,D,T).

testing(Id,Tick,D,T) ->
  case read() of
    Id -> mutex(Id,Tick,D,T);
    _ -> idle(Id,Tick,D,T)
  end.

mutex(Id,Tick,D,T) ->
  mce_erl:probe({enter,Id}),
  write(0),
  mce_erl:probe({exit,Id}),
  idle(Id,Tick,D,T).

read() ->
  case mcerlang:nget(id) of
    N when is_integer(N),N>=0 -> N
  end.

write(V) ->
  mcerlang:nput(id,V).

%% Support code

sleep(Milliseconds) ->
  receive after Milliseconds -> ok end.

latest(_Tick,0,F) ->
  mce_erl:urgent(0), F();
latest(Tick,Wait,F) ->
  mce_erl:urgent(0),
  mce_erl:choice
    ([fun () -> mce_erl:urgent(0), F() end,
      fun () ->
        mce_erl:urgent(0),
        receive after Tick -> latest(Tick,Wait-Tick,Fun) end
      end]).
```

Fig. 10: Fischer's mutual exclusion algorithm in Erlang

6.2 Expressive Power

As a final example we consider the verification of the nos_supervisor library [21]; this is a crucial software component used in several industrial projects at the LambdaStream company [27].

A supervisor is a process in charge of starting, stopping and monitoring a set of children (processes). Basically whenever a child process terminates the supervisor should restart it, i.e., spawn a new process executing the task of the terminated child. A supervisor typically supervises not only process workers, but also other supervisors, defining a hierarchical structure as shown in Fig. 11.

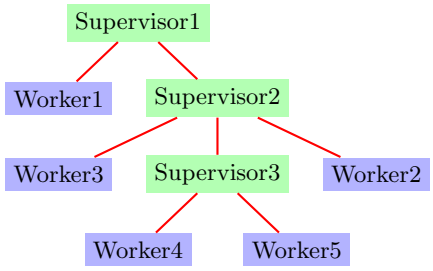

Fig. 11: A supervision tree.

The nos_supervisor is implemented in around 760 lines of Erlang code. To enable untimed verification of part of its functionality in [21] we had to modify the functions below:

```
%% Check if restarting a child again is admissible
add_restart(#child_spec{restart_intensity = infinity}) -> [];
add_restart(Spec=#child_spec{state = ChildState}) ->
  {MaxR,MaxT,Final} = Spec#child_spec.restart_intensity,
  Restarts = ChildState#child_state.restarts,
  check_restarts(MaxR, Final,
                 filter_restarts(MaxT, [now() | Restarts])).

%% Remove restarts older than MaxT
filter_restarts(MaxT, [H | Restarts]) ->
  F = fun(Restart) -> difference(Restart, H) < MaxT end,
  [H | lists:takewhile(F, Restarts)].

check_restarts(MaxR, Final, Restarts) ->
  case length(Restarts) > MaxR of
    true -> Final;
    false -> Restarts
  end.
```

These functions define the restarting policies of the supervisor. The function `add_restart` is called when a child process should be restarted due to having terminated abnormally. However, a constraint on restarting is that the child process may not have been restarted more than `MaxR` times within `MaxT` seconds. If this constraint is violated, the supervisor terminates all child processes and then itself.

Correctness properties are specified as safety monitors that inspect the actions of a the supervisor and processes interacting with the supervisor; see [21] for details. Using untimed McErlang we were not able to verify time dependent properties for the supervisor component, but had to resort to expressing the timing checks as a nondeterministic choice.

Using timed McErlang there is no need to change any of the 760 lines of code, although a few lines of code had to be added to delete the time clocks created using `nowRef()`. Of course we still need to create verification scenarios that explore the behaviour of the supervisor in detail, i.e., programming child process terminating abnormally and being restarted in narrow time intervals.

7 Conclusions

We have implemented a timed semantics for the Erlang programming language in the McErlang model checker, and have demonstrated that the resulting tool is capable of verifying timed systems. Compared to other similar semantics our semantics has a few interesting characteristics such as e.g. the absence of a dedicated time tick.

Currently the timed implementation is undergoing a study as to its suitability as a workbench for analysing Timed Rebeca programs. Earlier work [28] has implemented a translation from Timed Rebeca to Erlang, and has used untimed McErlang to simulate and test the resulting programs against correctness properties specified as safety monitors. In recent work, the new timed McErlang model checker, and the language extensions to specify urgency and clocks, are being used to verify Timed Rebeca programs.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. TCS **126** (1994) 183–235
2. Ouaknine, J.: Discrete analysis of continuous behaviour in real-time concurrent systems. PhD thesis, Oxford University (2001)
3. Moller, F., Tofts, C.M.N.: Behavioural abstraction in TCCS. In Kuich, W., ed.: ICALP. Volume 623 of Lecture Notes in Computer Science., Springer (1992)
4. Hansson, H., Jonsson, B.: A calculus for communicating systems with time and probabitilies. In: IEEE Real-Time Systems Symposium. (1990) 278–287
5. Léonard, L., Leduc, G.: A formal definition of time in LOTOS. Formal Asp. Comput. **10**(3) (1998) 248–266
6. Larsen, K.G., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT **1**(1-2) (1997)
7. Holzmann, G.J.: The model checker SPIN. IEEE Transactions on Software Engineering **23** (1997) 279–295

8. Lamport, L.: Real-time model checking is really simple. In Borrione, D., Paul, W.J., eds.: CHARME. Volume 3725 of LNCS., Springer (2005) 162–175
9. Wang, H., MacCaull, W.: Verifying real-time systems using explicit-time description methods. In Andova, S., McIver, A., D'Argenio, P.R., Cuijpers, P.J.L., Markovski, J., Morgan, C., Núñez, M., eds.: QFM. Volume 13 of EPTCS. (2009)
10. van den Berg, L., Strooper, P.A., Winter, K.: Introducing time in an industrial application of model-checking. In Leue, S., Merino, P., eds.: FMICS. Volume 4916 of LNCS., Springer (2007) 56–67
11. Fredlund, L.Å., Svensson, H.: McErlang: a model checker for a distributed functional programming language. In: Proceeding of the 12th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP), Freiburg, Germany, ACM (2007)
12. Guo, Q., Derrick, J., Hoch, C.: Verifying Erlang telecommunication systems with the process algebra muCRL. In Suzuki, K., Higashino, T., Yasumoto, K., El-Fakih, K., eds.: FORTE. Volume 5048 of LNCS., Springer (2008) 201–217
13. Guo, Q., Derrick, J.: Verification of timed Erlang/OTP components using the process algebra mucrl. In: Proceedings of the 2007 ACM SIGPLAN Workshop on Erlang, Freiburg, Germany, October 5, 2007. (2007) 55–64
14. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice-Hall (1996)
15. Cesarini, F., Thompson, S.: Erlang Programming – A Concurrent Approach to Software Development. O'Reilly Media (2009)
16. Blau, S., Rooth, J., Axell, J., Hellstrand, F., Buhrgard, M., Westin, T., Wicklund, G.: AXD 301: A new generation ATM switching system. Computer Networks **31**(6) (1999) 559–582
17. Wiger, U., Ask, G., Boortz, K.: World-class product certification using Erlang. SIGPLAN Not. **37** (December 2002) 25–34
18. McErlang: web page. https://babel.ls.fi.upm.es/trac/McErlang/ (April 2012)
19. Fredlund, L., Sánchez Penas, J.: Model checking a VoD server using McErlang. In: EUROCAST 2007. Volume 4739 of LNCS., Springer (2007) 539–546
20. Benac Earle, C., Fredlund, L., Iglesias, J., Ledezma, A.: Verifying robocup teams. Lecture Notes in Computer Science **5348/2009** (2009) 34–48
21. Castro, D., Gulías, V.M., Benac Earle, C., Fredlund, L.Å., Rivas, S.: A case study on verifying a supervisor component using McErlang. ENTCS. **271** (2011) 23–40
22. https://github.com/fredlund/McErlang-DTime (April 2012)
23. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. In Bernardo, M., Corradini, F., eds.: SFM. Volume 3185 of LNCS., Springer (2004) 200–236
24. Fredlund, L.Å.: A Framework for Reasoning about Erlang Code. PhD thesis, Royal Institute of Technology, Stockholm, Sweden (2001)
25. Svensson, H., Fredlund, L.Å.: A more accurate semantics for distributed Erlang. In: Proc. of the SIGPLAN workshop on Erlang, New York, USA, ACM (2007)
26. Gafni, E., Mitzenmacher, M.: Analysis of timing-based mutual exclusion with random times. In: In Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing, ACM Press (1999) 13–21
27. LambdaStream S.L.: web page. http://www.lambdastream.com/ (April 2012)
28. Aceto, L., Cimini, M., Ingólfsdóttir, A., Reynisson, A.H., Sigurdarson, S.H., Sirjani, M.: Modelling and simulation of asynchronous real-time systems using Timed Rebeca. In: FOCLASA. Volume 58 of EPTCS. (2011) 1–19